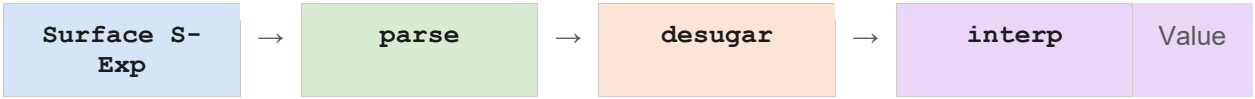
Language with Desugaring

Surface Language · Core Language · Desugarer · AST · Parser · Interpreter

Extends: Language with Functions & Closures · Covers pages 71–84

0. Big Picture: How SMoL Becomes Large

The key insight of this chapter is that a language can ***feel*** large to programmers while remaining ***small*** for implementors. Extra constructs — called **syntactic sugar** — are compiled away before interpretation. The program that removes sugar is the **desugarer**.



KEY The desugarer sits between the parser and the interpreter. The interpreter only ever sees the CORE language — it never needs to know about sugar constructs.

1. What Changes in This Version

Three things are added. Nothing from the previous version is removed or changed.

Component	Previous Version	This Version	Why
Grammar	6 forms	`8 forms (+and, +or)`	Sugar lives in surface only
`AST (Exp type)`	6 constructors	`8 constructors (+andE, +orE)`	Parser maps sugar → AST sugar nodes
Desugarer	`—  (did not exist)`	desugar : Exp → Exp	Translates andE/orE to ifE/letE
Interpreter	Handles 6 forms	`Now handles ifE too (core)`	if is core; and/or desugar into it
Runner	parse → interp	parse → desugar → interp	One new step inserted

2. Grammar (BNF)

We add three surface forms (shaded orange) and promote `if` to the core. Everything else is unchanged.

<expr>	**::=**	<num> numeric literal
		| { + <expr> <expr> } addition

	`| { if <expr> <expr> <expr> }`	conditional ◀ NOW CORE
	`| { lam <var> <expr> }`	anonymous function
	`| { let1 { <var> <expr> } <expr> }`	local binding
	`| { <expr> <expr> }`	function application
	`| { and <expr> <expr> }`	**sugar: both true ◀ NEW**
	`| { or <expr> <expr> }`	**sugar: either true ◀ NEW**
	`| <var>`	variable reference

3. AST — Updated Exp Type

Two new constructors are appended. The existing six constructors are **unchanged**. We also add `ifE` so the core language can represent conditionals.

Previous (unchanged)

```
(define-type Exp
  [numE  (n : Number)]
  [plusE (left : Exp)
         (right : Exp)]
  [varE  (name : Symbol)]
  [let1E (var  : Symbol)
         (val  : Exp)
         (body : Exp)]
  [lamE  (var  : Symbol)
         (body : Exp)]
  [appE  (fun  : Exp)
         (arg  : Exp)]
)
```

New constructors added ◀

```
  ; --- core conditional ---
  [ifE   (cnd  : Exp)   ; condition
         (thn  : Exp)   ; then-branch
         (els  : Exp)]  ; else-branch

  ; --- syntactic sugar ---
  [andE  (left : Exp)   ; {and e1 e2}
         (right : Exp)]

  [orE   (left : Exp)   ; {or e1 e2}
         (right : Exp)]
```

NOTE ifE is a CORE form that the interpreter handles directly. andE and orE are SUGAR — the desugarer translates them to ifE/let1E and they never reach the interpreter.

4. Desugaring Rules

Each sugar form has an equivalent expansion in the core language. This is what the desugarer produces:

4a. and → if

`{and e1 e2}` means: evaluate e1; if truthy, evaluate and return e2; otherwise return false.

Surface (sugar)	Core expansion

`{and e1 e2}`	`(ifE e1 e2 (numE 0))` if e1 is truthy → return e2; else → 0 (false)

4b. or → let1 + if (the safe version)

`{or e1 e2}` means: if e1 is truthy, return *its value* (not just true); otherwise return e2. We must bind e1 to avoid evaluating it twice:

Surface (sugar)	Core expansion

`{or e1 e2}`	`(let1E 'v e1`
	`  (ifE (varE 'v)`
	`  (varE 'v)`
	`  e2))`

WHY let? If we naively wrote (ifE e1 e1 e2), e1 would be evaluated TWICE — once for the test and once for the result. Binding to a fresh variable v ensures evaluation happens exactly once. This is the 'macro hazard' discussed on p.81.

5. Desugarer : Exp → Exp

The desugarer is a tree-walk that reconstructs the AST, translating sugar nodes and recursing into all sub-expressions. **It does not evaluate anything.**

```
;; desugar : Exp -> Exp
;; Translates surface AST nodes into core AST nodes.
;; Recurses into every sub-expression.
(define (desugar [e : Exp]) : Exp
  (type-case Exp e

    ; ── atoms: nothing to desugar ──────────────────────────
    [(numE n)         (numE n)]
    [(varE s)         (varE s)]

    ; ── recursive cases: desugar sub-expressions ───────────
    [(plusE l r)      (plusE  (desugar l) (desugar r))]
    [(ifE c t e)      (ifE    (desugar c) (desugar t) (desugar e))]
    [(lamE v b)       (lamE v (desugar b))]
    [(appE f a)       (appE   (desugar f) (desugar a))]
    [(let1E v val b) (let1E v (desugar val) (desugar b))]

    ; ── sugar: expand then recurse ─────────────────────────

    ; {and e1 e2} => (if e1 e2 false)
    [(andE l r)
     (desugar (ifE l r (numE 0)))]

    ; {or e1 e2} => (let1 v e1 (if v v e2))   <-- avoids double eval
```

```
    [(orE l r)
     (desugar
       (let1E 'v l
         (ifE (varE 'v)
              (varE 'v)
             r)))]))))
```

PATTERN Sugar cases call (desugar ...) on the expansion — not on the original sub-expressions. This means if e1 or e2 themselves contain sugar, it gets cleaned up in the same pass.

6. Parser : S-Exp → Exp (Updated)

Three new branches are added (highlighted). The parser maps surface syntax to AST — including sugar constructors. The desugarer, not the parser, decides what to do with them.

```
(define (parse [s : S-Exp]) : Exp
  (cond
    [(s-exp-number? s) (numE (s-exp->number s))]
    [(s-exp-symbol? s) (varE (s-exp->symbol s))]
    [(s-exp-list? s)
     (let ([l (s-exp->list s)])
       (cond
         ; {+ l r}
         [(symbol=? '+ (s-exp->symbol (first l)))
          (plusE (parse (second l)) (parse (third l)))]

         ; {if cond then else}                              ◀ NEW
         [(symbol=? 'if (s-exp->symbol (first l)))
          (ifE (parse (second l))
               (parse (third l))
               (parse (fourth l)))]

         ; {and e1 e2}  -- sugar                            ◀ NEW
         [(symbol=? 'and (s-exp->symbol (first l)))
          (andE (parse (second l)) (parse (third l)))]

         ; {or e1 e2}   -- sugar                            ◀ NEW
         [(symbol=? 'or (s-exp->symbol (first l)))
          (orE  (parse (second l)) (parse (third l)))]

         ; {lam x body}
         [(symbol=? 'lam (s-exp->symbol (first l)))
          (lamE (s-exp->symbol (second l))
                (parse (third l)))]

         ; {let1 {x val} body}
         [(symbol=? 'let1 (s-exp->symbol (first l)))
          (let ([binding (s-exp->list (second l))])
            (let1E (s-exp->symbol (first binding))
```

```
                    (parse (second binding))
                    (parse (third l)))))]

         ; {f arg}  -- anything else is application
         [else
          (appE (parse (first l)) (parse (second l)))]))]
    [else (error 'parse "not a number, symbol, or list")]))
```

7. Interpreter : Exp Env → Value (Updated)

One new case is added: `ifE`. The interpreter **never sees andE or orE** — they have already been eliminated by the desugarer.

```
(define (interp [e : Exp] [nv : Env]) : Value
  (type-case Exp e
    [(numE n)    (numV n)]
    [(varE s)    (lookup s nv)]
    [(plusE l r) (add (interp l nv) (interp r nv))]

    ; ── NEW: conditional ──────────────────────────────────────
    ; if the condition is NOT (numV 0) it is truthy
    [(ifE cnd thn els)
     (if (equal? (interp cnd nv) (numV 0))
         (interp els nv)
         (interp thn nv))]

    ; ── closures (unchanged) ──────────────────────────────────
    [(lamE v b)   (funV v b nv)]
    [(appE f a)
     (let ([fv (interp f nv)]
           [av (interp a nv)])
       (type-case Value fv
         [(funV v b nv*) (interp b (extend nv* v av))]
         [else (error 'app "not a function")]))]

    ; ── let1 (unchanged) ──────────────────────────────────────
    [(let1E var val body)
     (interp body (extend nv var (interp val nv)))]

    ; ── sugar nodes: should NEVER appear here ─────────────────
    [(andE _ _) (error 'interp "andE not desugared!")]
    [(orE  _ _) (error 'interp "orE not desugared!")]))
```

DESIGN	The andE/orE error cases are defensive programming. If the runner always calls desugar before interp, these branches are unreachable. They serve as a safety net during development.

8. Runner (Updated)

The only change is inserting the desugarer between parse and interp:

Previous runner

```
(define (run [s : S-Exp]) :
Value
  (interp (parse s)
          empty-env))
```

Updated runner ◀

```
(define (run [s : S-Exp]) :
Value
  (interp (desugar (parse s))
          empty-env))
; parse → desugar → interp
```

9. Helper: add (unchanged)

```
;; add : Value Value -> Value
(define (add [l : Value] [r : Value]) : Value
  (cond
    [(and (numV? l) (numV? r))
     (numV (+ (numV-n l) (numV-n r)))]
    [else (error 'add "not a number")]))
```

10. Tests

All previous tests continue to pass. New tests exercise the sugar forms:

```
(print-only-errors #true)

; ── all previous tests (still pass) ──────────────────────────
(test (run '3)               (numV 3))
(test (run '{+ 3 4})         (numV 7))
(test (run '{let1 {x 3} x})  (numV 3))
(test (run '{{lam x {+ x x}} 3}) (numV 6))

; ── new: if (core) ───────────────────────────────────────────
(test (run '{if 1 42 99})    (numV 42))  ; truthy  -> then
(test (run '{if 0 42 99})    (numV 99))  ; 0=false -> else
(test (run '{if {+ 1 2} 7 8}) (numV 7)); non-zero -> truthy

; ── new: and (sugar) ─────────────────────────────────────────
(test (run '{and 1 2})       (numV 2))   ; both truthy -> right value
(test (run '{and 0 2})       (numV 0))   ; left=0/false -> 0
(test (run '{and 1 0})       (numV 0))   ; right=0      -> 0

; ── new: or (sugar) ──────────────────────────────────────────
(test (run '{or 3 99})       (numV 3))   ; left truthy  -> left VALUE
(test (run '{or 0 99})       (numV 99))  ; left=0       -> right
(test (run '{or 0 0})        (numV 0))   ; both false   -> 0
```

```
; ── or preserves value, not just truthiness ──────────────────
(test (run '{let1 {x 5}
                {or x 99}})
      (numV 5))    ; returns 5, not just 'true'

; ── nested sugar ─────────────────────────────────────────────
(test (run '{and {or 0 3} {+ 1 1}}) (numV 2))

; ── sugar inside lambda ──────────────────────────────────────
(test (run '{let1 {f {lam x {or x 10}}}
                {f 0}})
      (numV 10))
```

11. Pipeline Trace — {or 0 {+ 1 2}}

Step-by-step through parse → desugar → interp for a concrete or expression:

Step 1 · run is called

```
(run '{or 0 {+ 1 2}})
```

Step 2 · parse builds the AST

```
head = 'or  =>  orE branch
  left  = parse(0)          => (numE 0)
  right = parse({+ 1 2})    => (plusE (numE 1) (numE 2))

parse output:
  (orE (numE 0) (plusE (numE 1) (numE 2)))
```

Step 3 · desugar transforms the tree

```
Input:  (orE (numE 0) (plusE (numE 1) (numE 2)))

orE case fires:
  l = (numE 0)
  r = (plusE (numE 1) (numE 2))

Expansion:
  (desugar
    (let1E 'v (numE 0)
      (ifE (varE 'v)
           (varE 'v)
           (plusE (numE 1) (numE 2)))))

Recurse: all sub-nodes are atoms/core => no further sugar

desugar output:
  (let1E 'v (numE 0)
```

```
    (ifE (varE 'v) (varE 'v) (plusE (numE 1) (numE 2))))
```

Step 4 · interp evaluates the core AST

```
interp (let1E 'v (numE 0) (ifE ...)) empty-env

  let1E fires:
    val-result = interp (numE 0) empty-env = (numV 0)
    new-env    = { v -> (numV 0) }
    interp (ifE (varE 'v) (varE 'v) (plusE...)) new-env

  ifE fires:
    cnd = interp (varE 'v) { v->(numV 0) } = (numV 0)
    (equal? (numV 0) (numV 0))  =>  #true
    => take ELSE branch
    interp (plusE (numE 1) (numE 2)) new-env

  plusE fires:
    LEFT  = interp (numE 1) env = (numV 1)
    RIGHT = interp (numE 2) env = (numV 2)
    add: (numV (+ 1 2)) = (numV 3)

Final Output: (numV 3)
```

12. Key Concepts

Concept	Code / Location	What it means
Syntactic sugar	andE, orE in AST	Convenient forms absent from core; interpreter never sees them
Core language	numE,plusE,ifE,varE,lamE,…	The minimal set the interpreter handles directly
Desugarer	desugar : Exp → Exp	Tree-walk that eliminates sugar before interpretation
Double-eval hazard	ifE e1 e1 e2	Duplicating a sub-expression evaluates it twice — use let to bind first
Fresh variable	'v in orE expansion	Binding to a new name prevents capture of existing variable 'v in scope
`Hygiene (Racket)`	macro expander colors	Names introduced by macros are kept separate from user-defined names
Pipeline order	parse → desugar → interp	Desugaring always runs after parse, always before interp

13. Surface Language vs Core Language

Surface Language (what programmers write)

All 8 forms:

- `num, +, if, lam, let1, app, var`
- `and` ◀ `sugar`
- `or` ◀ `sugar`

Core Language (what the interpreter sees)

6 forms only:

- `num`
- `+`
- `if` (was sugar, now core)
- `lam`
- `let1`
- `app`
- `var`